INL/CON-07-12089
PREPRINT

# Streamlining of the RELAP5-3D Code

## NURETH-12

George L. Mesina
Joshua M. Hykes
Donna Post Guillen

November 2007

The INL is a
U.S. Department of Energy
National Laboratory
operated by
Battelle Energy Alliance

# STREAMLINING OF THE RELAP5-3D CODE

**George L. Mesina[1], Joshua M. Hykes[2] and Donna Post Guillen[1]**
[1]Idaho National Laboratory, P.O. Box 1625, Idaho Falls, ID 83415
P.O. Box 1625, Idaho Falls, ID 83415
George.Mesina@inl.gov; Donna.Guillen@inl.gov

[2]Department of Nuclear Engineering
Pennsylvania State University
0355 Atherton Hall, University Park, PA 16802
Jmh539@psu.edu

## ABSTRACT

Many successful and long-lived computer programs undergo physical modeling improvements and optimizations over a period of years. These have often been incorporated into the programs without regard for readability or runtime and have created maintenance and development difficulties. Thus, modifying a program to incorporate new models, upgrade numerics, optimize to new hardware, and adapt to new platforms becomes more difficult as the code grows larger and more complex. Conversely, the probability of introducing errors is significantly reduced when the code is subsequently streamlined with a focus on maintainability and readability. For a code such as RELAP5-3D, with numerous past, present. and planned future developments, it is valuable to modify the code infrastructure so that the modifications are incorporated in a consistent and reliable manner. For these reasons, the process of streamlining RELAP5-3D by a combination of restructuring and rewriting the code has been undertaken. Results of this work are reported here.

**KEYWORDS**
RELAP5, restructure, thermal hydraulics

## 1. INTRODUCTION

Legacy codes are applications which have been developed over the course of many years, even several decades. Examples of such codes in the nuclear industry include RELAP5-3D [1], TRACE [2], CATHARE [3], RETRAN-03 [4], COBRA-IV [5], CONTAIN [6], MELCOR [7], and MCNP [8] to name but a few. It is common for these applications to have had multiple programmers who have both modified parts of the program and added coding to incorporate new features. This results in disparate coding structures and styles throughout the program that complicates maintenance and development tasks. Factors that contribute to this are:

- Modification of existing features to extend their range of application.
- Improvements to code numerics for greater accuracy and robustness.
- Development of new features with fundamentally different database structures.
- Importation and incorporation of library subprograms and/or full programs.
- Use of conditional code to protect/incorporate features for specific purposes (e.g., different platforms, different customer groups, and proprietary code purposes).
- Adaptations of the code to various operating systems and compilers.
- Optimizations to take advantage of computer architectures (such as RISC, vector, parallel, and message passing).
- A mixture of the different programming styles from developers of all these factors resulting in the lack of a consistent or cohesive style.

Often, coding introduced to implement one or more these factors complicates the subprogram unit into which it is placed. It can disrupt the flow of the original algorithm with jumps, such as GOTO statements and returns, or with conditional sections of code. Depending upon the skill of the programmers and constraints such as time and funding, this has also resulted in redundant coding (sections of repeated code that could have been a subprogram) and even unused sections of code. In addition, the development of many legacy codes started before modern standards of software engineering were established; consequently, the codes often contain coding styles inconsistent with today's programming paradigms.

The cumulative effect of years of changes to a legacy code is that the code may lack the structure of newer codes. This has led to a need for streamlining of some legacy codes. Streamlining of a procedural program includes:

- Reorganization of the code into the structured programming paradigm.
- Removal of unused and obsolete sections of code.
- Simplification of complex subprograms.
- Establishment and application of consistent programming style rules.
- Removal of programming workarounds (“tricks”) imposed by language limitations

A good example of this is the RELAP5 code, a state-of-the-art nuclear power plant safety analysis program developed by the Idaho National Laboratory (INL). Designed to analyze operational transients and a variety of accident scenarios for pressurized light water reactors, RELAP5 has been extended in many ways over the years to increase its capabilities, as well as provide updates, corrections, and enhancements. These extensions include modeling of boiling water, molten salt, liquid-metal, and gas-cooled fission reactors as well as fusion reactors. The code was also extended to drive nuclear plant operator training simulators. Some changes that

brought about these extensions include the addition of 26 fluids, multi-dimensional thermal-hydraulics and multi-dimensional neutron kinetics, improved numerics, and code optimizations for speed, platforms, and programming paradigms (such as shared memory multi-processing).

As listed above, the development effort has involved a variety of programming short-cuts and tricks, code imports, and the use of numerous code developers of varying ability and styles. The resulting code is difficult to work with because of readability and understandability issues.

In general, the process of software quality assurance verifies and validates a program version. This guarantees that the code works correctly by solving the equations right (verification) and solving the right equations (validation) [9]. However, this process does nothing to improve readability and understandability of the database and code. To improve the coding, the computer industry has developed a number of code analyzers and processors. These include complexity analyzers, coverage analyzers, optimization analyzers (parallel region analysis, for example), and reformatting and restructuring software. Applying these tools can lead a programmer to a variety of ideas/suggestions to improve a computer program.

Complexity analysis of RELAP5-3D revealed that part of the readability issue was too many logic flow paths within its larger subroutines. Additionally, coding was written for different compilers and by different authors in different styles. The largest difficulty with understanding the coding, apart from these readability issues, is the complex database or container array.

To address these issues and make the code easier to maintain and develop, streamlining of RELAP5-3D has been undertaken. The database has been completely reconstructed and rewritten as Fortran 90 modules. The coding has been reorganized into the structured programming paradigm. Streamlining the source code has many aspects and is the subject of this paper; the reconstruction of the database will be the subject of another paper. The result of streamlining, both source code and database, is a code that is easier to read, understand, develop, and maintain.

## 2. CODE DEVELOPMENT AND EFFECTS

### Partial History of RELAP5 Code Developments

The RELAP5 code was originally designed to analyze large break Loss of Coolant Accidents (LOCA) in a Pressurized Water Reactor (PWR). Therefore, great effort was given to the time scales and pressures consistent with these operating and accident conditions. Therefore, a semi-implicit time-stepping scheme subject to the material Courant limit and steam tables operating in the range of atmospheric pressure to 22 MPa for light water were developed.

Almost from the beginning, the development team decided to expand the original design to analyze small break LOCAs as well. They also included the capability to analyze the Boiling Water Reactor (BWR) as it operates in the same time scale and pressure ranges. Later, it was straightforward to add heavy water as another coolant because processes to develop the tables for heavy and light water fluid properties were similar.

Originally, RELAP5 solved a set of five governing partial differential equations: two phasic continuity equations, two phasic momentum equations, and one energy equation, and achieved closure of the system of equations by assuming the minor phase was saturated. There was a major change to six governing equations in RELAP5/MOD2, wherein the algebraic energy equation was replaced by a second phasic energy conservation equation. This immensely complicated several subroutines, especially the equation of state calculations which now had to account for metastable states in both phases.

It was later deemed valuable to take larger time steps both for slowly changing operational transients and for steady-state calculations. A new numerical time-stepping algorithm was developed, the so-called nearly-implicit method. This method added a number of new subroutines, added complexity to many existing subroutines and to the overall solution algorithm in general.

Rigorous assessment of many mostly empirical hydrodynamic and heat transfer correlations and their replacement with conventional “text book” correlations led to a great deal of development in the form of new subroutines and modification of existing ones. So extensive were the changes, the new code version was named RELAP5/MOD3. At the same time, a much improved off-take model and a Counter-Current Flow Limiting (CCFL) model were added.

A very large change was the development of multidimensional physics for use in modeling asymmetric accident scenarios (such as a cold-leg break in a multi-loop plant) more accurately for the heavy water cooled tritium production reactors at Savannah River. This affected the database, which previously held primarily linear arrays. The multidimensional hydrodynamics introduced new kinds of data, new input and output, over half a dozen new large subroutines, and extensive modifications to several others. The NESTLE [10] multidimensional nodal kinetics program was imported by creating a large interface array to contain its data, means to access the data and coding, and new input. The BPLU [11] linear equation solver was introduced to reduce the run time for large multidimensional models, particularly on vector and parallel platforms. All these went into RELAP5/MOD 3.2 as conditional coding in the early 1990s.

A Department of Energy (DOE) Cooperative Research and Development Agreement (CRADA) was undertaken with Scientific Applications International Corporation (SAIC) in 1994 to develop a real-time version of RELAP5 for use in operator training simulators. The work was completed in 1996 and real-time RELAP5 has been placed on numerous training simulators throughout the world.

The most extensive change of the CRADA project was the parallelization of the RELAP5 code. The target platform was a 233 MHz, 2-CPU, shared memory platform on which a sufficiently detailed model of a reactor was required to run at simulator speed, namely completion of one time step every 0.1 seconds of wall-clock time. The parallel coding changes were introduced via calls to the Kuck and Associates Incorporated (KAI) parallel library procedures. The processing was divided among four threads, one of which was the master and the others its slaves. The implementation not only changed the fundamental way the processing took place, but was implemented as CRADA-protected conditional code; therefore, the normal single-processor serial processing method had to continue to work.

The CRADA produced RELAP5-RT, the real-time version of RELAP5; it could be configured from RELAP5/MOD3.2 by choosing the correct set of conditional coding. The CRADA also directly led to RELAP5-3D which is configured by selecting the multidimensional hydrodynamics and neutron kinetics.

Graphical User Interfaces (GUI) have been added to RELAP5 to display calculated data. The Nuclear Plant Analyzer (NPA) [12] and SIMPORT [13] visualize data on a 2D schematic of the plant with displays resembling actual nuclear plant controls. The RELAP5-3D Graphical User Interfaces (RGUI) [14] displays plant conditions on a 3D representation of the plant. All three require interface coding, both SIMPORT, now renamed 3KeyMaster by Western Services Corp., and RGUI are interactive, and RGUI adds another set of data to RELAP5-3D.

To model severe code damage, the Severe Code Damage Analysis Program (SCDAP) [15] was incorporated into RELAP5/MOD2 for analyzing severe accidents, such as Three Mile Island. It has its own database that is completely different from that of RELAP5-3D as conditional code. Both NESTLE and BPLU had been standalone programs comprised of suites of subroutines that each had totally different databases than RELAP5 and were imported directly into the code with little change beyond imposition of certain style conventions.

Rather than continuing to subsume entire programs into RELAP5-3D, a method to couple RELAP5-3D to programs with features useful to analyzing specific plants was developed. The PVMEXEC [16] program is a master program that uses the Parallel Virtual Machine (PVM) message passing methodology to communicate between RELAP5-3D and other programs, such as FLUENT, to model large single-phase regions, or to CONTAIN to model the containment in a nuclear power plant. The conditional coding within RELAP5-3D that communicates with the PVMEXEC program makes use of the RELAP5-3D database but extends it in ways different from all previous database extensions.

## 2.2 Recent and ongoing developments

Part of code maintenance is adapting to new hardware, operating systems, and operating system software, such as compilers. From its inception, RELAP5-3D has been adapted to various computing platforms ranging from vector-parallel to RISC workstations to PCs and from the Cray OS to various vendor versions of Unix and Linux to MS Windows. It was written in Fortran 66, converted to Fortran 77, and is undergoing transformation to Fortran 90. This latter development reduces machine dependency by replacing platform specific coding with Fortran 90 intrinsics, thus eliminating some conditional coding.

Current and future planned developments include continual physics improvements for the Next Generation Nuclear Plant (NGNP) and for the Global Nuclear Energy Program (GNEP). The former will be a gas cooled reactor [17], while the latter will be sodium cooled. Also, there is a continuing need to develop the code for existing plants, as well as in the design and analysis of Generation 3+ plants, such as AP1000, ABWR, ESBWR, USEPR, and USAPWR.

Fluids data continue to be added to enable the modeling of different reactor designs. Fluids have been added and/or enhanced recently for the analysis of space reactors and Generation 4 conceptual designs. For example, helium, xenon, and helium-xenon data were added for space reactors, carbon dioxide for supercritical $CO_2$ conceptual design, lead-bismuth for a fast liquid-metal generation four design, and four fluorine-based molten salts were added for a variant of the NGNP [18]. Such additions are incorporated into RELAP5-3D merely by adding a fluid property table generator and small coding changes to allow the user to select them via input.

Some specific recent developments for Generation 3 and 4 plants include the Henry-Fauske critical flow model, and heat transfer correlations such as Gneilinksi for gas-cooled reactor applications and Bishop and Oka-Koshizuka for supercritical water applications [1]. These changes have relatively minor effect on the database and the coding effects are confined to either a new subroutine or subroutine section. Another current and proprietary development is the addition of conditional coding for carrying out Appendix K calculations. This also has a relatively minor impact on the database, but requires introduction of conditional code to a large number of subroutines.

The result of all this code development has been a code that works correctly but has become unwieldy to develop. RELAP5-3D matches separate effects and plant performance data and, when configured as RELAP5-RT, provides excellent performance on nuclear plant operator training simulators in terms of speed and actual plant reaction to operator actions. However, the coding is difficult to read, understand and maintain. The partial development history of RELAP5-3D recounted in Section 2 demonstrates that every code complication factor listed in Section 1 is in effect. Steps are being taken to overcome maintenance and development issues introduced by these factors. The process is program streamlining.

## 3. STREAMLINING

Program streamlining addresses two separate areas, namely the database and the coding. This paper addresses only the coding area. For a procedural code, such as RELAP5-3D, the greatest portion of program streamlining is to reorganize the code into the structured programming paradigm as described in Section 3.1. A software tool can be employed to restructure code; however, the tools have limitations as discussed in Section 3.2. These limitations can be overcome through pre- and post-processing development for the subroutines being restructured, which is explained in further detail in Section 3.3. Automatic tools are not always the best solution; some subprograms are so complex that manual reworking is the best approach, as outlined in Section 3.4. Finally, measurements of code readability and understandability improvement are presented in Section 3.5.

### 3.1 Structured Programming

Many attempts have been made, throughout the history of computing, to produce code that is easy to read and understand because, in general, such programming is more reliable and the processes of maintenance, debugging, and development are simpler and less costly. In the 1960's and 1970's, the concept of structured programming for procedural, as opposed to object-oriented, programs was developed. It arose from the structured programming theorem. It states that three

ways of combining programs—sequencing, selection, and iteration—are sufficient to express any computable function [19]. From this Dijkstra published his famous article, "GOTO Statement Considered Harmful" [20] wherein he laid the foundation for structured programming.

Dijkstra's structured programming paradigm can be summarized as follows [21]. Structured programming is a technique for organizing and coding computer programs in which a hierarchy of "modules" is used, each having a single entry and a single exit point, and in which control is passed downward through the structure with no unconditional branches to higher levels of the structure. There are three types of flow control: sequential; test (if and case); and iteration (loop). The term "block of code" or simply block is used in place of "module" to avoid confusion for languages with "module" constructs, such as Fortran 90.

Not every computer scientist agreed that each block must have exactly one exit point and some argued that GOTO can be useful in some situations [22], [23]. Along with structured programming came a number of related principles, such as top-down design, stepwise refinement, modularity, and data encapsulation. All of these ideas have proven their worth and remain a part of software practice today [24].

Accordingly, the principles of structured programming were applied to the restructuring of RELAP5-3D as follows: Structured programs can be broken into sub-sections or blocks, each with one point of entry and at least one point of exit, such that control passes downward from one block to the next with no unconditional branches to higher levels of the structure.

A very important feature of restructuring is that it can be undertaken subprogram by subprogram, as each is independent of every other one. Thus, properly planned, restructuring can be carried out in a manner that does not interfere with other maintenance and development work.

### 3.2 Software Tool and Limitations

After a study was made of various restructuring tools available, FOR_STRUCT [25] was selected based on performance on some very complex RELAP5-3D subroutines. This product can reorganize FORTRAN 77 programming into code that satisfies the working definition of structured coding. It can also apply some simple formatting rules as it rewrites the code. In the process of restructuring, FOR_STRUCT will make specific improvements such as:

- Restructure loops with backwards GOTOs into do loops.
- Replace GOTOs with equivalent structured constructs where it makes sense to do so (i.e., not indiscriminately).
- Remove/renumber statement labels consistently.
- Convert do/continue constructs to the more modern do/endo form.
- Identify unused blocks of code and unused variables.
- Consistently apply user-supplied programming style rules.

However, the restructuring of RELAP5-3D subroutines is made complex by several issues. A prime rule for restructuring is that its input (a program unit) must compile with no errors. Therefore, pre-compiler directives cannot be present nor can any coding outside the ANSI FORTRAN standard. Another important issue is sheer size of the input. Really large, complex subprograms are not fully restructured by FOR_STRUCT. A seemingly lesser limitation is that not all comments move correctly with the block of coding to which they belong. This affects the strategies available in overcoming the first three limitations. Finally, not all restructured coding is more readable than the original. In these cases, manual work is needed to improve readability. Because the restructuring tools have limitations, they cannot be applied directly. Strategies for overcoming these limitations are discussed in Section 3.3.

## 3.3 Restructuring Algorithm

The limitations of the restructuring tools require some sort of preprocessing to prepare a subprogram as proper input to the restructuring program. After restructuring, the modifications must be undone to ensure all features, such as pre-compiler directives, are intact. Preprocessing and post-processing can be largely automated. However for several reasons, manual processing for long or complex subprograms is sometimes necessary prior to automated preprocessing and is often necessary after automated post-processing. This is summarized in Figure 1.

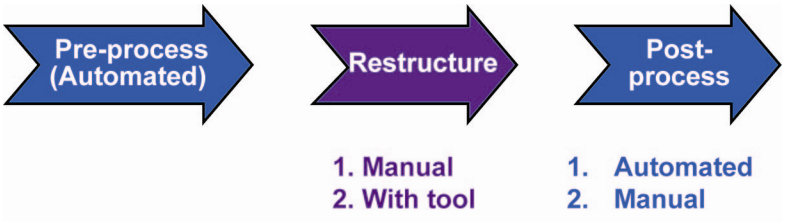


**Figure 1 High Level Restructuring Algorithm.**

Each limitation is examined in turn and methods implemented to overcome it are explained.

### 3.3.1 Non-standard executable code

The existence of non-standard executable code prevents restructuring using FOR_STRUCT. Most FORTRAN restructuring tools, including FOR_STRUCT, accept only ANSI-standard FORTRAN 77. Non-standard code falls into two categories, machine-specific coding and coding from Fortran 90 or a higher ANSI standard. An example of the former is the Cray-specific buffer statement.

In most cases, where the non-standard coding is not within a decision statement, the entire statement can be commented out and restructuring works correctly. For a decision a legitimate Fortran 77 item replaces the non-standard item, such as a Fortran 90 derived type reference. The replacements are recorded for reverse substitution after restructuring.

### 3.3.2 Conditional Code

The existence of conditional code prevents restructuring using FOR_STRUCT. Conditional code in RELAP5-3D is marked by pre-processor directives #ifdef, #ifndef, #else, and #endif, where #ifdef and #ifndef are each followed by a single argument, the pre-processor flag. Coding between a #ifdef and its paired #endif is included in the pre-processor output if the flag is activated by its inclusion in a define file. Otherwise the coding in the #ifndef or #else section, if it exists, is output by the pre-processor.

Pre-compiler directives are not part of the standard. The program unit will conform to the standard if the directives are somehow removed. Either converting them to comments or applying the pre-processor will remove them, but neither of these is sufficient.

Simply commenting out directives can lead to compiler errors. Some examples are: multiple occurrences of a statement number from different conditional blocks of code; successive do-loops statements (before any end do) with the same do-index; improper nesting; and multiple declaration of the same variable.

Alternatively, applying the pre-compiler eliminates the directives. It also removes some conditional coding. The missing coding must be restored to the restructured subprogram. One difficulty with this is finding the precise location to place correctly the missing coding when the rest of the code may be completely rearranged. Another difficulty is that the missing code is not restructured. Finally, the missing code may contain jumps (GOTO statements, read and open condition branch labels) to statement labels that have been eliminated or renumbered.

The solution is to combine the two approaches. First, create a comment that records where each directive is, but place it on the outside of the conditional block of coding. The comment serves as a marker of where the removed coding must be returned. Second, create one define file for each possible combination of pre-compiler flags, and then apply the pre-compiler to each. With N different flags there are $2^N$ combinations, resulting in $2^N$ pre-compiled files. Third, restructure each pre-compiled subprogram producing $2^N$ output files. Finally, starting with the first file, replace markers with restructured blocks from another output file until all markers are gone and restore the directives from the markers. This is summarized in Figure 2.

In fact, $2^N$ define files are not needed because many combinations would produce exactly the same pre-compiled file. In practice, only the minimal number of combinations is employed.

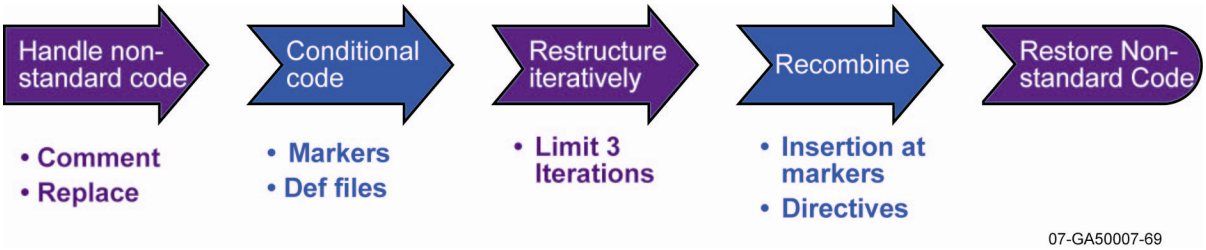


**Figure 2 Flowchart for handling conditional code**

Everything discussed thus far has been fully automated, except restoration of directives. The tool associates comments with block of code immediately below them. Thus, #endif comment markers can be misplaced. Misplaced #endif directives must be properly relocated manually.

### 3.3.3 Long program units

Experience shows that FOR_STRUCT does not fully restructure program units with more than about 500 lines of code. Such subroutines are improved in measurable ways by the restructuring program, for example, there are fewer GO TO statements, but more restructuring is possible.

The algorithm for the automated work is summarized in Figure 3.

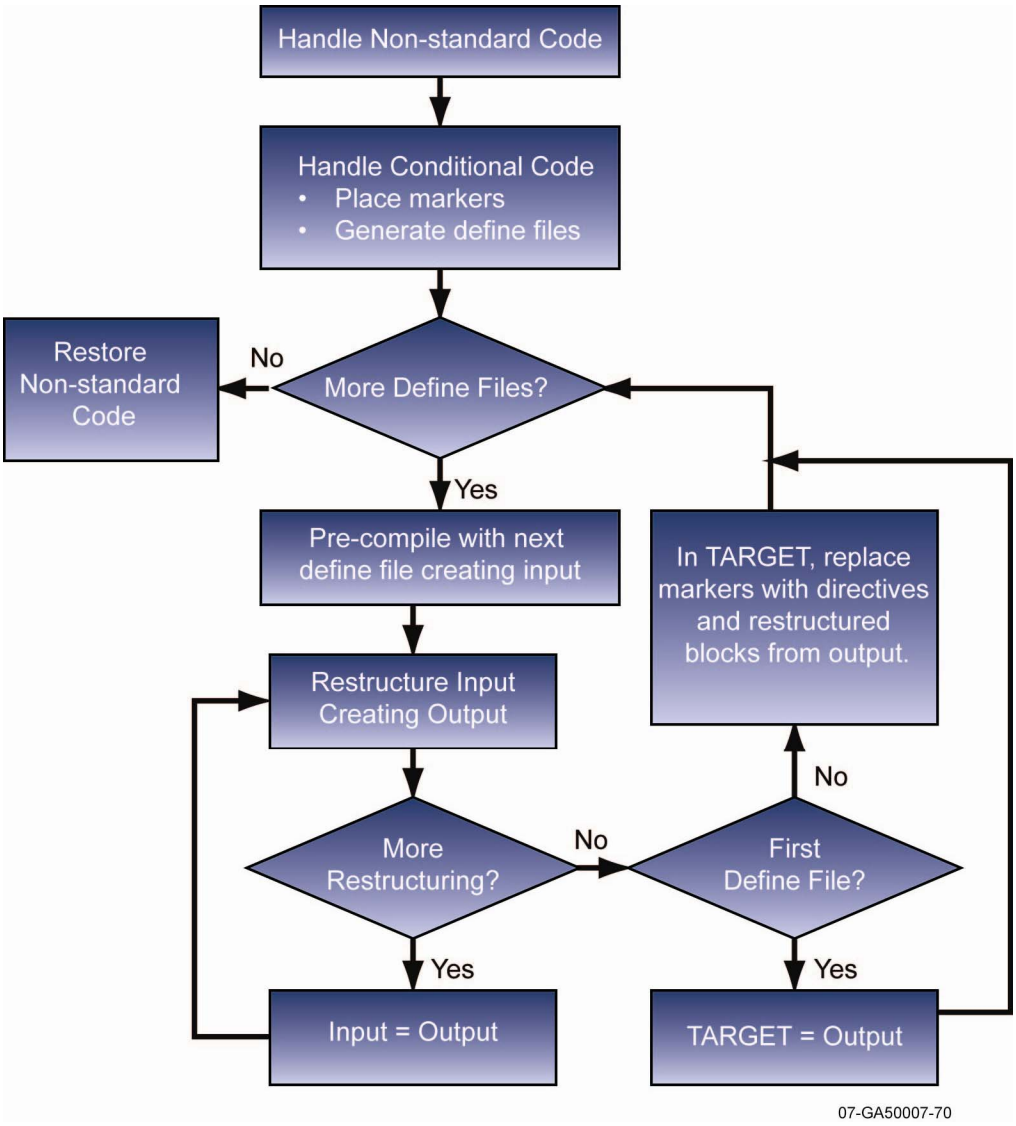


**Figure 3 Restructuring Algorithm – automated portion.**

The solution is to apply restructuring iteratively by applying the restructuring tool to its own output file. With each successive iteration, incremental improvement is made. Experience

showed that there was usually little or no gain after three iterations and this was the upper limit applied to RELAP5-3D subroutines. In cases where three iterations left a significant measure of structuring undone, manual creation of contained subroutines was applied, Section 3.4.1. Note that iteration is performed before post-processing.

## 3.4 Restructuring via rewriting

It must be noted that not all code is more legible or completely restructured after application of Algorithm 3.3.3. The true goal of streamlining is code that runs correctly and is easier to read and maintain. In some cases, the modified code satisfies the structured program definition, but it is actually harder to read. The modified code may have extremely deep indentation that causes too many continuation lines. Hard-to-comprehend “do while” constructs may have replaced backward GOTO statements. Numerous GO TO statements may remain and new ones may have been introduced. In these cases, some manual rewriting is necessary to increase readability.

For subroutines with a significant number of misplaced #endif directives (Section 3.3.2) and for very large and complex subroutines, manual restructuring can be applied before using the restructuring tool. At its highest level, this is done by creating contained subroutines. For subroutines that the tool cannot improve or that have grown difficult to develop and maintain because of a combination of the factors listed in Section 1, a complete rewrite is in order.

### 3.4.1 Internal subroutines

Blocks of code that perform a specific function in the high-level algorithm of the subroutine are identified. The original subroutine is modified so that this block of code has only one entrance and one exit. This can be a difficult process for a block with many jump exits and many statement labels that serve as entry points. The block of code is then moved to the contain section of the original subroutine and a call statement is left in its place.

A block of code marked by a directive pair (#ifdef or #ifndef and #endif) is easy to identify. If it is one where the #endif becomes misplaced, placing the directives inside the contained subroutine prevents misplacing the #endif. Also, if a particular sequence of coding is repeated many times with different variables, turning that into a contained subroutine both shortens and simplifies the code.

Manually moving blocks of coding to contained subroutines both restructures and enhances readability. Creating internal subroutines can be done before or after automated restructuring.

### 3.4.2 Complete rewriting

An example of a RELAP5-3D subroutine that required rewriting is SCNREQ. Three tools were applied to restructure it. They not only failed to improve the code, but actually added GO TO statements and increased its length. Moreover, the “restructured” code was less legible.

Rewriting requires less time than was devoted to developing the original subroutine. Development time includes the time for creating the original program unit and the time for

incorporating new features and debugging and maintaining it. The complete set of functions that an existing subroutine performs is known at rewrite time.

Rewriting a subprogram begins with recording all its functions and analyzing its underlying algorithm. An efficient means of implementing all the functions is identified and written as a new algorithm. This algorithm is converted to source code according to the established style guidelines for the program. It is then debugged and fully tested.

This procedure was applied to subroutine SCNREQ, whose original purpose was to provide the index in the container array of a specific datum from among specific data types requested by the user via input. This had been expanded to optionally provide conversion factors, physical units, and to include many more data types of differing structures. GO TO, computed GO TO, and arithmetic if statements provided all transfers. Manual rewriting produced a subroutine, IREQUEST, with the same functionality in only 1600 lines, not 2900, and has no GO TO statements or statement labels except for formats. It is very easy to read and modify due to its use of contained subroutines, adherence to the program guidelines, and single programming style.

### 3.5 Metrics

For streamlining to be successful, the code must become more legible in measurable ways. Some of measurable items that make a code difficult to read and understand are:

- Large number of GO TO statements
- Backward GO TO statements
- Many statement labels
- Repeated, redundant and dead code

These items were addressed by the restructuring tool and by manual streamlining in Section 3.4. Of these, only two sections of dead code were identified and removed. Measures of the other three items are given in Tables I and II. Both compare RELAP5-3D version 2.4.1, denoted as "Before," and 2.5.5 with SCNREQ replaced by IREQUEST, denoted "After".

**Table I. Effect of streamlining on GO TO statements**

| **Go to statements** | Before | After | Ratio |
|---|---|---|---|
| Total files | 554 | 610 | |
| Total subprograms | 588 | 714 | |
| 0 GO TOs | 255 | 386 | 0.66 |
| 10 or more GO TOs | 138 | 93 | 1.48 |
| 25 or more GO TOs | 66 | 45 | 1.47 |
| 100 or more GO TOs | 9 | 1 | 9.0 |
| Total GO TOs | 6707 | 3977 | 1.69 |
| **Computed GO TOs** | 100 | 7 | 14.29 |
| **Backward GO TOs** | 822 | 125 | 6.32 |
| Avg./file | 12.1 | 6.7 | 1.81 |
| Avg./subpgm. | 12.0 | 6.1 | 1.97 |
| Maximum | 778 | 157 | 4.96 |

As can be seen, in Table I, the number of GO TO statements was reduced significantly. The number of files with no GO TO statements increased by 130 files (over 50%), the maximum number of GO TO statements in any file was reduced by nearly a factor of five, while the average number of GO TO statements per subprogram was nearly halved.

Most significant was the reduction in backward GO TO statements, which cause the logic paths to become tangled and make subprograms harder to decipher. Backward GO TO statements were reduced by more than a factor of six. Also, computed GO TO statements, which can be converted to either CASE or multiple-branch IF statements, were reduced by greater than 14 times, and in fact, nearly eliminated. The only remaining computed GO TO statements reside in subroutines destined to be manually rewritten.

**Table II. Effect of streamlining on statement labels**

| **Statement labels** | Before | After | Ratio |
|---|---|---|---|
| 0 labels | 99 | 273 | 0.36 |
| 10 or more labels | 239 | 139 | 1.72 |
| 25 or more labels | 112 | 68 | 1.65 |
| 100 or more labels | 29 | 14 | 2.07 |
| Total labels | 12328 | 7005 | 1.76 |
| Labels of formats | 3751 | 3203 | 1.17 |
| Non-format labels | 8577 | 3802 | 2.26 |
| NFL Avg./file | 15.48 | 6.23 | 2.48 |
| NFL Avg./subprogram | 14.59 | 5.32 | 2.74 |
| Maximum | 594 | 258 | 2.30 |

Statement labels serve to mark the targets of GO TO statements, target of condition flags on read statements, final statements of DO loops, and FORMAT statements of I/O statements. Reducing the number of targets makes a subprogram easier to decipher and understand, although format labels have less impact on readability because they do not influence logic flow paths. In Table II, NFL stands for Non-Format Labels.

The reduction in statement labels was significant. The number of files with 100 or more labels was more than halved. Files with no labels increased by a factor of 2.76. The total number of non-format labels decreased by a factor of 2.26. Most significantly, the average non-format labels per subprogram decreased by more than a factor of 2.74.

Finally, another important aspect of streamlining is application of a uniform style. A uniform style was imposed by the FOR_STRUCT tool. Thus 100% of the files restructured with FOR_STRUCT now conform to certain coding style rules.

### 3.6 Other Considerations

Some questions someone might want to ask before embarking upon a streamlining project with a legacy code are the following:

- What does it cost?
- Does it change the length of the source code?
- What effect does it have on code portability?
- What effect does it have on code runtime?

To make it worthwhile, it must be an effort that does not take very long and does not impact the maintenance and development schedule. Ideally, it should be something that pays for itself over time. We completed the work in 26 weeks, although that was spread over 14 months as most of the work was carried out during two summers by laboratory intern and co-author Joshua Hykes.

Another consideration is the effect it has on code size. Normally, the number of lines remains about the same or increases slightly when, for example, multiple returns are replaced by a single point of return and GO TO statements that jump to it. Use of internal subroutines to preprocess the file before applying the restructuring tool also adds a few lines for the following statements: call, contains, and the internal subroutine's return, end and header statements. However, removal of dead coding, manual elimination of obsolete coding, and rewriting of subroutines can significantly reduce the number of lines of code in individual subroutines.

Another consideration is code portability. Often the manual operations weed out machine specific coding for old machines and operating systems that can be replaced by more modern language constructs. The replacement of obsolescent features, such as arithmetic and computed GO TO statements, improves portability and code longevity; there will come a time when an ANSI Fortran standard deletes the currently obsolescent features.

Finally, code run time was not noticeably altered in any way. This was difficult to test as other developments were ongoing simultaneously; however, when tested with just the streamlining updates, before and after timings were virtually identical.

## 4. CONCLUSIONS

Streamlining is a process whose goal is to make a program more readable and understandable to reduce maintenance and development costs. For legacy codes, this process becomes increasingly necessary with age, size, and complexity. The process of streamlining the source code was explained and illustrated with flowcharts. Streamlining was applied to RELAP5-3D and measurements of the improvements were shown. With a reasonable amount of effort and without affecting code calculations or runtime, significant improvements in readability were achieved.

## ACKNOWLEDGMENTS

This work was funded by the International RELAP5 Users Group and authored by Battelle Energy Alliance, LLC under Contract No. DE-AC07-05ID14517 with the U.S. DOE.

## REFERENCES

1. RELAP5-3D Code Development Team, "RELAP5-3D Code Manual," *INEEL-EXT-98-00834 Revision 2.4*, Idaho National Laboratory, Jun, (2005).

2. J. W. Spore, et al., "TRAC-M/Fortran 90 (Version 3.0) Theory Manual," *NUREG/CR-6724*, Los Alamos National Laboratory, Jul, (2001).
3. D. Bestion, G. Geffraye, "The CATHARE Code," *DTP/SMTH/LMDS/EM/2001-063*, Laboratoire de Modelization Diphasique et des Simulateurs, CEA, Apr, (2002).
4. J. H. McFadden, et al., "RETRAN-03 Code Manual," *EPRI NP-7450*, May, (1992).
5. C. W. Stewart, et al, "COBRA-IV: The Model and the Method," *BNWL-2214*, Pacific Northwest Laboratory, (1997).
6. K. W. Washington, et al., "Reference Manual for the CONTAIN 1.1 Code for Containment Severe Accident Analysis," *NUREG/CR5715*, *SAND91-0835*, Sandia National Laboratory, May, (1991).
7. R. O. Gauntt, et al., "*MELCOR Computer Code Manuals*," *NUREG/CR-6119, SAND2001-0979P*, Sandia National Laboratory, Apr, (2001).
8. X-5 Monte Carlo Team, "MCNP5 Monte Carlo N-Particle Transport Code System", *LA-UR-03-1987*, Los Alamo National Laboratory, Apr, (2003).
9. P. J. Roache, "Verification and Validation in Computational Science and Engineering," Hermos Publishers, NM, pp 23, (1998).
10. P. J. Turinsky, et al., "NESTLE: A Few-Group Neutron Diffusion Equation Solver Utilizing the Nodal Expansion Method for Eigenvalue, Adjoint, Fixed-Source Steady-State and Transient Problems," *EGG-NRE-11406*, Idaho National Engineering Laboratory, Jun, (1994).
11. G. L. Mesina, " Border-Profile LU Solver for RELAP5-3D," *Proeedings. of the 1998 RELAP5 International Users Seminar*, College Station, Texas, May 17-21, (1998).
12. D. M. Snider, K. L. Wagner, W. H. Grush, and K. R. Jones, "Nuclear Plant Analyzer," *NUREG/CR-6291*, *INEL-94/0123*, Idaho National Engineering Laboratory, Jan, (1995).
13. SIMPORT, https://www.ws-corp.com/wsc/presentations/3key/3keymaster.asp (2007).
14. J. A. Galbraith, and G. L. Mesina, "RELAP5/RGUI Architectural Framework," *Proceedings of the 8th International Conference on Nuclear Energy*, Baltimore, MD, USA, Apr 2-6 (2000).
15. SCDAP/RELAP5-3D Development Team, "SCDAP/ RELAP5-3D Code Manuals," *INEEL-EXT-02-00589*, Revision 2.2, Idaho National Engineering Laboratory, Oct, (2003).
16. W., L. Weaver, "The Application Programming Interface for the PVMEXEC Program and Associated Code Coupling System PVMEXEC," http://www.inl.gov/relap5/pvm_api.pdf, *INL/EXT-05-00107*, Idaho National Engineering Laboratory, Mar, (2003).
17. C. B. Davis, T. D. Marshall, K. D. Weaver, "Modeling the GFR with RELAP5-3D," *Proceedings of the 2005 RELAP5 International Users Seminar*, Jackson Hole, WY, Sep, 7-9, (2005).
18. C. B. Davis, "Implementation of Molten Salt Properties into RELAP5-3D/ATHENA, *INEEL/EXT-05-02658*, Idaho National Engineering and Environmental Laboratory, Jan, (2005).
19. C. Boehm and G. Jacopini, "Flow Figures, Turing Machines and Languages with only Two Formation Rules", *Comunications. of the ACM*, **9** (5), pp 366-371, (1966).
20. E. W. Dijkstra, "GOTO Statement Considered Harmful," *Communications of the ACM,* **11**, (3), pp. 147-8, (1968).
21. General Services Administration, "The Definition of Structured Programming," *Federal Standard 1037C*, Telecom Glossary, (2000).
22. D. Knuth. "Structured Programming with go to Statements," Computing Surveys, **6** (4), pp 261-301, (1974).

23. F. Rubin, "'GOTO Considered Harmful' Considered Harmful" (letter to the editor), *Communications of the ACM*, **30**, (3), pp. 195-196, (1987).
24. B. Hayes, "The Post-OOP Paradigm", *American Scientist* **91** (2), pp 106-110, (2003).
25. Cobalt Blue, Inc., *FOR_STRUCT®, Your FORTRAN Structuring Solution*, 11585 Jones Bridge Rd, Suite #420-306, Alpharetta, GA, 30005, USA, (1997).